Formal Methods for High Integrity GPU Software Development and Verification

Dimitris Aspetakis* Leonidas Kosmidis*† Matina Maria Trompouki*† Jose Ruiz‡ Gabor Marosy§

*Barcelona Supercomputing Center (BSC), Spain
†Universitat Politècnica de Catalunya (UPC), Spain
‡AdaCore, France
§European Space Agency (ESA), The Netherlands

***Abstract*—Modern safety critical systems require high levels of performance for advanced functionalities, which are not possible with the simple conventional architectures currently used in them. Embedded General Purpose Graphics Processing Units (GPGPUs) are among the hardware technologies which can provide the high performance required in these domains. However, their massively parallel nature complicates the verification of their software and increases its cost because it usually involves code coverage through extensive human-driven testing.**

The Ada SPARK language has traditionally been used in highly-critical environments for its formal verification capabilities and powerful type system. The use of such tools which are backed up by theorem provers, has significantly lowered the amount of effort needed to validate functionality of safety-critical systems.

In this European Space Agency (ESA) funded project, we utilize AdaCore's CUDA backend for Ada in conjunction with the SPARK language subset to assess the state of static verification for GPU kernels. We assess the error detection capabilities of the available tools and we formulate a methodology to maximise their effectiveness. Moreover, our project results using ESA's open source GPU4S Benchmarking suite show that common programming mistakes in GPU software development can be prevented.

***Index Terms*—Ada SPARK, Embedded GPGPUs, Formal Verification, Safety Critical Systems**

I. Introduction

As more and more safety-critical systems utilize computers to make decisions, the need for safe, automated software verification methodologies is slowly becoming a necessity. Automated verification tools were developed to aid the resource-heavy manual testing and code reviews. Those verification tools can be divided into two subcategories: dynamic and static. Dynamic tools are usually easy to apply, but their effectiveness relies on the tests' coverage exhaustiveness. Test coverage is not an easy problem to solve, and even if done properly, it might end up needing more computation time than what is acceptable. Moreover, dynamic solutions can only detect defects but cannot prove their absence. Static verification tools on the other hand, operate on the semantics of systems, trying to prove properties that hold for all possible program states.

Modern safety critical systems require high performance processing power for the implementation of advanced functionalities (e.g. advanced driver assistance systems — ADAS). Some of these features, like automatic emergency breaking, are mandatory for all vehicles sold in the European Union starting from 2022 [1]. Traditional processing elements used in safety critical systems can't keep up with the rates of innovation the industry dictates. Similar requirements are found in other domains like aerospace, where there is a need for more autonomy such as in planetary exploration through rovers or deep space missions, where real-time communication and telecommand is not possible.

Embedded General Purpose Graphics Processing Units are the most promising candidates for such systems due to their high performance, low power consumption and easier programmability than competing parallel systems. For this reason, their use is considered across all types of safety critical systems (automotive [2], avionics [3], [4] and space [5]).

While conventional software systems running on CPUs have long enjoyed the advantages of static verification approaches [6], [7], GPUs have not really dived deep into this area. Some verification tools do exist in academia, but their research and prototype nature has kept them from extensive industry adoption.

In this work, we take advantage of AdaCore's SPARK verification toolchain, with track record used in industry across several safety critical domains for the last decades, for single core software verification. In particular, we use the under-development Ada backend for CUDA, and evaluate how they can be combined to formally verify safety critical GPGPU software. We try to identify current limitations and provide guidelines and best-practices to maximise the error detection capability. Our work specifically addresses the CPU-to-GPU communication parts of a codebase, a part typically overlooked by previous efforts.

By applying our formulated programming methodology to ESA's open source benchmarking GPU4S Bench [8], we show that common programming mistakes in GPU software development can be prevented.

II. Background and Relater Work

A. General Purpose GPUs

GPUs (graphics processing units) are specialized hardware devices introduced to accelerate the (highly data-parallel) graphics in our computing systems. Their SIMT (single instruction, multiple threads) model got the attention of the HPC

979-8-3503-4859-0/DATE24/© 2024 EDAA

(high-performance computing) industry, leading to the introduction of general purpose GPU programming. Nowadays, GPUs hold a significant role in HPC and the industry, being most appropriate for massively parallel workloads.

GPUs are programmed using heterogeneous programming models such as CUDA – used in NVIDIA devices – and OpenCL, a Khronos standard supporting multiple vendors. As accelerators, they require the presence of a host processor (CPU). CPUs and GPUs have distinct address spaces, even in the case where both systems share the same main memory, as it is the case in embedded GPUs. It is the programmer's responsibility to manually perform memory transfers between them, before and after a kernel is executed.

The CPU's code portion is in charge of performing the main interaction with the system and heavy parallel computations are offloaded to GPUs. This happens in the form of *kernels*, which are functions sent to GPUs for execution. Programmers need to specify a kernel *configuration* describing how many threads will be used for the kernel execution, as well as how these threads are organised in *blocks* within the *Grid* of threads. Kernels can communicate through a fast on-chip memory known as *shared memory* and synchronise their execution using *barriers*. Threads from different blocks can only communicate through *Global Memory* (DRAM). Finally, GPU threads are executed in lockstep groups of (typically) 32 threads, known as *warps*.

B. Formal Methods

Formal methods are techniques based on mathematical principles, that aid in the specification, design and verification of systems. Their results can be trusted, and no additional proofs should be needed for their promises, given that the design and implementation of the tools itself is correct.

Formal methods traditionally involved knowledge of Hoare and separation logic, as well as the use of niche programming languages and proof assistants [9]. Program properties needed to be described mathematically, in order to prove either that the program adheres to its specification, or that the generated executable code is equivalent to what the programmer specified [10] [11].

This rare expertise has prevented widespread use of formal methods despite their benefits. However, advances in programming languages, such as the ones provided by Ada SPARK and Frama-C [12], allow their use by non-experts without prior knowledge of advanced logic methods.

C. Ada SPARK

Ada is a programming language, first released in 1980. It has been heavily used by the safety-critical sector, since it supports both an extensive pool of safety checks at runtime, and the Design-by-Contract (DbC) methodology [13]. SPARK is a formally defined subset of Ada. It provides many advantages, specifically important for high-integrity, safety critical systems: flow analysis, proof of functional correctness, of safety and security properties, of termination, and proof of absence of run-time exceptions.

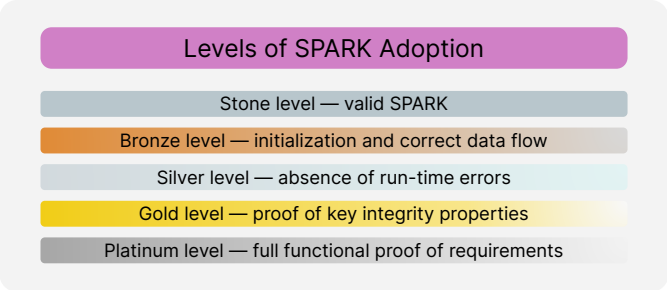

Fig. 1. SPARK adoption levels

SPARK's DbC methodology allows the introduction of specification statements about the software behaviour such as pre/post-conditions and invariants. These statements are not executable, but are converted internally to logic statements known as Verification Conditions (VC) used to prove program properties. In fact, the AdaCore compiler generates additional VCs from the executable code, which are combined with the specification statements. These VCs are passed to an *automatic theorem prover system* which either proves that they all hold, or finds a counter example in which some of them are not satisfied. In this case, a user-friendly message is provided to the programmer pointing out the problem, with possible hints at additional information needed in the specification in order to make it hold.

SPARK defines 5 adoption levels: Stone, Bronze, Silver, Gold and Platinum (Fig. 1). Stone level is achieved when the code is converted to the executable subset of SPARK. Bronze level guarantees correct initialisation and control flow. In Silver level, the absence of runtime errors is ensured. In Gold level key integrity properties of the software are proven correct. Finally, in Platinum level there is a full functional proof of requirements, i.e. that the software meets its specification. Increasing the SPARK adoption level for a given program provides higher assurance while requiring more effort, and depends on the software criticality.

AdaCore has recently released a CUDA backend for its Ada compiler (GNAT), which can generate code for NVIDIA GPUs [14]. Our main project contribution was to examine how SPARK's formal prover can be used in conjunction with this new backend, and identify possible directions for tighter integration of the tools.

D. Related Work

There have been a handful of academic tools developed for statically detecting potential coding errors in GPU kernels. `GPU_Verify` [15], `PUG` [16] and `GKLEE` [17] can be used to prove data race freedom (DRF) and synchronization errors. `VerCors` [18] and `Vericuda` [19] utilize annotations to prove functional correctness on top of DRF. `ESBMC-GPU` [20] and `CIVL` [21] are the most complete examples of formal verification on GPGPUs. They are able to detect all sorts of runtime errors, like division-by-zero, index-out-of-bounds

```
1  -- Type declarations
2  type Vector is array (Natural range <>) of Float;
3  type Vector_Host_Access is access Vector;
4  type Vector_Device_Access is access Vector
5   with Designated_Storage_Model =>
6     CUDA.Storage_Models.Model;
7
8  -- Host code for Main
9  procedure Main with SPARK_Mode is
10    Vector_Size : constant Positive := 2**18;
11
12    A, B, C, C_Host : Vector (0 .. Vector_Size - 1);
13
14    Threads_Per_Block : Pos3 := (2**8, 1, 1);
15    Blocks_Per_Grid : Pos3 :=
16       ((Vector_Size + Threads_Per_Block.X - 1) /
17         Threads_Per_Block.X, 1, 1);
18 begin
19    -- Generate vectors A and B
20    (...)
21
22    VectorAddWrapper (Threads_Per_Block,
23                      Blocks_Per_Grid, A, B, C);
24 end Main;
```

Listing 1. Types and Main (host code)

accesses, integer overflows, user-specified assertions, as well as synchronization errors.

Unfortunately, most of these tools work only for specific older versions of CUDA and are not actively maintained, neither they are developed and verified with the rigor required for use in safety critical systems. As a consequence, the adoption of the aforementioned efforts in industry still remains low, since safety-critical domains usually require support of strict standards and qualified tools.

MISRA C [7] compliance checkers and the Ada SPARK toolchain [6] are two examples of verified tools used extensively in real-world applications. However, they have only been applied to CPU programs as of today. In this project, we evaluate the applicability of Ada's SPARK verification tools[1] in the GPGPU programming space.

III. GPGPU Software Verification Methodology in Ada SPARK

The CUDA backend is an on-going project for AdaCore, and hence it's not integrated with the SPARK toolchain yet. As such, one of our project's contributions was to find out how to make use of SPARK tools with Ada's CUDA interface. Due to lacking integration, we are forced to ignore some warnings and carefully avoid analysing certain procedures to circumvent unavoidable errors.

In this section we provide an overview of the methodology we have developed for the verification of GPGPU software written in Ada SPARK, using a simple example from our public repository [22].

As stated earlier, writing code for GPUs entails the kernels themselves (device i.e. GPU code) and their invocation (host code), shown in code listings 2 and 1 respectively. Having no significant work specific to the CUDA backend in SPARK yet, the parts we need to circumvent are (1) heap allocations with CUDA's storage model attribute, (2) the kernel invocation itself and (3) calls from the CUDA API.

```
1  -- Kernel specification
2  procedure Vector_Add (A, B, C : Vector_Device_Access)
3   with CUDA_Global;
4
5  -- Cuda indexing function
6  function Cuda_Index
7   (B_Dim, B_Idx, T_Idx : unsigned) return Natural with
8   SPARK_Mode => Off is
9  begin
10   return Natural (Block_Dim * Block_Idx + Thread_Idx);
11 end Cuda_Index;
12
13 -- Kernel body
14 procedure VectorAdd
15  (A, B, C : Vector_Device_Access)
16 is
17   -- Mirror wrapper's preconditions with assumptions:
18   X : Natural := Cuda_Index (Block_Dim.X,
19                              Block_Idx.X,
20                              Thread_Idx.X);
21   pragma Assume (A'First = 0 and B'First = 0 and
22                  C'First = 0);
23   pragma Assume (A'Last = B'Last and then
24                  B'Last = C'Last);
25   pragma Assume (A'Last <= Natural'Last - 31);
26   Max_X : Natural := ((A'Last + 31) / 32) * 32;
27   pragma Assume (X in 0 .. Max_X);
28 begin
29   if X <= A'Last then
30      C (X) := A (X) + B (X);
31   end if;
32 end VectorAdd;
```

Listing 2. Kernel (device code)

For (1), we simply pretend that the device access type has the default storage model by removing lines 5–6 from Listing 1 every time we run the prover.

For (3), our only CUDA-specific invocation is during the construction of our kernel's index. Because the `Block_Dim`, `Block_Idx` and `Thread_Idx` API functions return an unsigned integer from Ada's C interface, the conversion to our vectors' index type (that is, Ada's `Natural` type) is unsafe, let alone the arithmetic over/under-flows that might arise, as the prover is eager to point out.

Given the CUDA language specification, the kernel's call dimensions cannot exceed 32 bit integers, and therefore cannot exceed the upper bound of Ada's `Natural` type. Hence we use a function that will not get analyzed for the conversion (Listing 2, lines 5–11).

The last thing we must attend to is (2), the kernel invocation. Practically, we need to assure that the constructed index never points outside our vectors' bounds. This comes from a lack of semantics passed from host code to device code.

In fact, one of the most challenging problems for the application of formal methods in GPUs is to make sure that not only the CPU and the GPU code is analysed by the prover in isolation, but information is exchanged between them.

Therefore, we need a way to define the relationship of our kernel's input vectors and the indexing procedure. To combat this issue, we propose the following three-stage pattern:

[1]AdaCore's tools are qualified for use in numerous safety critical environments. While the under development CUDA backend we use in this project is not qualified yet, we believe that it will be qualified once its development is completed, following AdaCore's long tradition in all safety critical domains.

1) Construct a wrapper for the CUDA kernel invocation and the data transfers before and after it (Listing 3). Importantly, the wrapper's parameters include both the input/output vectors and the desired CUDA block and grid dimensions. The body of the wrapper **will not get analyzed** for SPARK verification.
2) Add preconditions in the wrapper's specification that dictate invariants among the vectors' ranges and the given CUDA block and grid dimensions. The wrapper's specification **will get analyzed** for SPARK verification.
3) In the declaration part of our kernel's body (Listing 2), reflect the wrapper's preconditions with Ada assumptions to properly inform the prover. Due to the underlying GPU architecture, we might get more threads than we expect. Specifically, the dimensions of our CUDA index will probably get rounded up to a multiple of the architecture's warp size (in our case, 32), and hence we need one more Ada assumption for this. This way, both the specification and body of our kernel **will get analyzed** for SPARK verification.

The best way to apply this pattern in practice, is to make any assumptions in the declaration part of the kernel's body you need to achieve verification, and then mechanically reflect them at the wrapper's specification with preconditions.

It is worth mentioning how both the `Cuda_Index` function and the wrapper's body have a straightforward and mechanical implementation, and therefore can be automated. That is important, since those are the two entities that do not get analyzed for SPARK verification.

The safety of the proposed pattern is robust. Should there is a mistake in the kernel's specification preconditions, leaving room for buffer overflow errors, any potential buffer overflow will get reported on the code written inside the kernel's body. Should there is a mistake in the host code, like giving incorrect dimensions to the wrapper, the error will get reported on the wrapper's preconditions.

By ensuring the prover's semantics' communication between the host and device code and therefore providing consistency between them, we are able to use the same verification properties for both host (CPU) and device (GPU) code. We have developed a series of GPGPU code examples in which we intentionally inject certain types of errors, whom we are later able to identify through the SPARK tools.

In particular, we are able to prove the absence of the following runtime errors in GPGPU software: buffer overflows in GPU code or in the interaction between host and device (eg. memory transfers and copies), integer overflow and underflow, division by 0, use of unitialised variables and floating point range errors. In addition, we are able to prove the correctness of a GPU kernel, i.e. that it conforms to its specification though preconditions, postconditions and invariants using *ghost procedures*, i.e. non executable procedures which are used by the prover and describe the kernel behaviour. This allows reaching gold and platinum adoption level, e.g. not only proving that the code is correct (i.e. doesn't have runtime errors) but also conforms to its requirements (i.e. implements the correct functionality). All our examples are available at [22].

```
1  -- Wrapper specification
2  procedure VectorAddWrapper
3   (Threads_Per_Block, Blocks_Per_Grid : Pos3;
4    A, B : Vector; C : out Vector) with
5   SPARK_Mode => On,
6   Pre =>
7    Threads_Per_Block.X * Blocks_Per_Grid.X in
8      Positive'Range
9    and then
10   (A'First = 0 and B'First = 0 and C'First = 0 and
11    A'Last =
12      Threads_Per_Block.X * Blocks_Per_Grid.X - 1 and
13    B'Last =
14      Threads_Per_Block.X * Blocks_Per_Grid.X - 1 and
15    C'Last =
16      Threads_Per_Block.X * Blocks_Per_Grid.X - 1);
17
18 -- Wrapper body
19 procedure VectorAddWrapper
20  (Threads_Per_Block, Blocks_Per_Grid : Pos3;
21   A, B : Vector; C : out Vector) with
22   SPARK_Mode => Off
23 is
24   D_A : Vector_Device_Access := new Vector (A'Range);
25   D_B : Vector_Device_Access := new Vector (B'Range);
26   D_C : Vector_Device_Access := new Vector (C'Range);
27 begin
28   D_A.all := A;
29   D_B.all := B;
30
31   pragma Cuda_Execute (VectorAdd (D_A, D_B, D_C),
32      Blocks_Per_Grid, Threads_Per_Block);
33
34   C := D_C.all;
35 end VectorAddWrapper;
```

Listing 3. Kernel Wrapper (host code)

Note however, that due to the incomplete status of the Ada backend for CUDA and the accompanying SPARK tools which do not currently support CUDA's shared memory and thread synchronisation constructs, it is not possible to identify programming mistakes related to them. Since we are unable to make use of those features, they do not pose a risk to be found in Ada SPARK GPGPU software. As soon as these features are implemented in the Ada SPARK backend, we will develop appropriate methodologies for their detection through SPARK tools, too.

IV. Use Cases

In addition to the artificially constructed examples mentioned above, we also implemented use cases with actual GPGPU software for safety critical systems. In particular, we ported European Space Agency's open source GPU4S Bench benchmarking suite [23] to Ada SPARK. GPU4S Bench has been developed within the GPU4S (GPU for Space) ESA funded project [5], and it is representative of computationally intensive algorithmic building blocks used in multiple domains of the aerospace sector. Our implementation is released as open source, in the GPU4S Ada SPARK repository [24].

Note that the GPU4S Bench belongs to ESA's open source OBPMark benchmarking suite [25], which was developed in the GPU4S project. It contains 3 types of benchmarks:

- GPU4S Bench: algorithmic building blocks commonly found in space software. It is also known as OBPMark Kernels. The selection of the algorithms was performed based on a survey on space software across all divisions of Airbus Defence and Space [23].
- OBPMark: full space applications. A collection of representative on-board software such as image processing [26], synthetic aperture radar (SAR), CCSDS 121 and 122 compression [27] and CCSDS encryption.
- OBPMark-ML: machine learning space applications. Two machine learning applications representing possible on-board data processing software. A cloud screening application and a ship detection application.

All OBPMark benchmarking suites are co-hosted at [28] and they are open source with a GPL-compatible ESA license.

We ported the GPU4S benchmark kernels to Ada SPARK, achieving Bronze level SPARK adoption. As a first step, we ported them to Ada SPARK without any annotations, ensuring that they are only using the Ada SPARK subset. Next, we applied the methodology we developed and described in previous section. Until now, we are able to prove correct initialisation and control flow (bronze level), as well as absence of runtime errors (silver level) using the 3-stage wrapper pattern. In the future, we are going to apply more steps, in order to reach a higher SPARK adoption level.

The ported benchmarks include kernels for matrix multiplication, max pooling, relu and softmax activation functions, a FIR and a 2D correlation filter, a wavelet transform, softmax, and finally a Local Response Normalization (LRN) kernel.

As expected, all ported benchmarks work properly and provide identical results with their CUDA counterparts. However, apart from the formal proof of their absence of runtime errors we did not have anything to compare with.

After our porting was completed, in the context of another project [29] we have detected two defects in the C version of the FIR benchmark of the GPU4S Bench. That version was the basis of the CUDA and OpenCL code, and therefore the GPU code had the same defects. The first problem was related to unitialised memory, since one of the arrays was not fully initialised, due to wrong size specification which didn't match its allocation size. The second problem was an out-of-bound access in the FIR function, due to wrong size specification in the call.

Interestingly, the defect was not creating a functional error such as corruption or failure of the output verification which is embedded in GPU4S benchmarks, although it was tested in multiple different processors (CPUs: x86, ARM, SPARC, NOEL-V, GPUs: NVIDIA, AMD, ARM) and under multiple operating systems (Linux, RTEMS). However, the error did manifested only when running on the GR740 (Frontgrade Gaisler's LEON4 space processor) under the RTEMS SMP space qualified real-time operating system.

Examining again the Ada SPARK port of the FIR benchmark, neither of the two defects were present, showing the effectiveness of protection level offered by the language as well as its formal tools.

Therefore our work demonstrates that porting actual GPU programming software from CUDA to Ada SPARK's GPU backend is feasible, and opens the door of increasing the assurance of GPU software.

V. Conclusions

In the "Formal Methods for GPU Software Development and Verification" ESA-funded activity, we are exploring the use of formal methods in GPU software, using AdaCore's (currently under development) Ada backend for CUDA.

We constructed a set of test programs to show how CUDA kernels can be programmed in the Ada SPARK subset, examining several types of common programming mistakes arising in GPU programming, and showing how these can be prevented. We have noticed that GPU programming in Ada is not very different from CUDA, although its syntax is more verbose. We observed that even the use of Ada alone can prevent common GPU programming mistakes, and we have shown that GPU programming in the SPARK subset of the language is also possible and further helps in preventing GPU programming pitfalls.

Moreover, we have explored some of the formal verification capabilities offered by the SPARK tooschain, in order to prove the absence of certain errors in GPU code. However, we have noticed that since the CUDA backend is currently under development and has loose integration with the SPARK formal tools, the detection of certain classes of errors is not possible at this point. This includes the incorrect use of shared memory, data races and wrong thread synchronisation within a kernel, as well as consistency checks between the CPU and GPU code.

To mitigate the last limitation, we came up with a programming pattern which can allow the use of SPARK's formal checks to verify the consistency between CPU and GPU code, and detect any possible buffer overflow errors — a major issue when formal methods are used in GPUs. Following our proposed pattern, programmers can have verification guarantees for their GPU code at a level on-par with conventional SPARK verification on CPUs.

As part of the project, we have ported a space-relevant open source GPU benchmarking suite, GPU4S Bench [23], achieving bronze-level SPARK verification, following the methodologies proposed in this deliverable. This way, we show that they are applicable in realistic safety-critical GPU applications. In particular, we have proven that our code is free of runtime errors, while the original code had some later discovered defects which were unnoticed for long time.

This confirms why testing alone is not enough and that formal methods can help finding hard to find errors.

All the Ada SPARK code developments of the project are released as open source [22] [24], contributing to the wider adoption of Ada SPARK and specifically for GPU development and verification.

VI. Future Work

Our work in this project will be extended in order to cover cases which we have identified that can be automated. This includes the implementation of the 3-stage programming pattern we proposed in Section III. In particular, we are currently developing a GPU source to source code translator, which can relieve the programmer from this mechanical task, and ensure that it is implemented without introducing any programming mistake in the process.

Moreover, as new features will become available by AdaCore, such as support for shared memory or synchronization in CUDA kernels, we will explore the effectiveness of formal SPARK tools to detect related mistakes.

Acknowledgments

This work was funded by the European Space Agency (ESA) activity "Formal Methods for GPU Software Development and Verification" (ESA STAR AO 2-1856/22/NL/GLC/ov). The authors thank AdaCore for donation of a license and early access to Ada's CUDA backend compiler along with their verification toolchain, as well as invaluable technical support. This work was also partially supported by the Spanish Ministry of Economy and Competitiveness under grants PID2019-107255GB-C21 and IJC-2020-045931-I (Spanish State Research Agency / Agencia Española de Investigación (AEI) / http://dx.doi.org/10.13039/501100011033) and by the Department of Research and Universities of the Government of Catalonia with a grant to the CAOS Research Group (Code: 2021 SGR 00637).

References

[1] P. E. Ross, "Brakes that Slam Themselves: Automatic emergency braking will become standard in Europe," *IEEE Spectrum*, vol. 59, no. 1, 2022.

[2] C. Q. Peralta, M. M. Trompouki, and L. Kosmidis, "Evaluation of SYCL's Suitability for High-Performance Critical Systems," in *Proceedings of the International Workshop on OpenCL (IWOCL)*, 2023.

[3] M. Benito, M. M. Trompouki, L. Kosmidis, J. D. Garcia, S. Carretero, and K. Wenger, "Comparison of GPU Computing Methodologies for Safety-Critical Systems: An Avionics Case Study," in *Design, Automation & Test in Europe Conference (DATE)*, 2021.

[4] M. Maria Trompouki and L. Kosmidis, "DO-178C Certification of General-Purpose GPU Software: Review of Existing Methods and Future Directions," in *Digital Avionics Systems Conference (DASC)*, 2021.

[5] L. Kosmidis, I. Rodriguez, A. Jover-Alvarez, S. Alcaide, J. Lachaize, O. Notebaert, A. Certain, and D. Steenari, "GPU4S: Major Project Outcomes, Lessons Learnt and Way Forward," in *Design, Automation & Test in Europe Conference (DATE)*, 2021.

[6] B. Carré and J. Garnsworthy, "SPARK—an Annotated Ada Subset for Safety-Critical Programming," in *Proceedings of the Conference on TRI-ADA '90*, 1990.

[7] R. Bagnara, A. Bagnara, and P. M. Hill, "The MISRA C Coding Standard and its Role in the Development and Analysis of Safety- and Security-Critical Embedded Software," 2018.

[8] L. Kosmidis, I. Rodriguez, Álvaro Jover, S. Alcaide, J. Lachaize, J. Abella, O. Notebaert, F. J. Cazorla, and D. Steenari, "GPU4S: Embedded GPUs in Space - Latest Project Updates," *Microprocessors and Microsystems*, vol. 77, 2020.

[9] A. W. Appel, R. Dockins, A. Hobor, L. Beringer, J. Dodds, G. Stewart, S. Blazy, and X. Leroy, *Program Logics for Certified Compilers*. Cambridge University Press, 2014.

[10] X. Leroy, "Formal Verification of a Realistic Compiler," *Commun. ACM*, vol. 52, no. 7, 2009.

[11] ——, "Formal Verification of an Optimizing Compiler," in *2007 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, 2007.

[12] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-C," in *International Conference on Software Engineering and Formal Methods*, 2012.

[13] B. Meyer, "Applying 'Design by Contract'," *Computer*, vol. 25, no. 10, 1992.

[14] AdaCore, "GNAT for CUDA repository," https://github.com/AdaCore/cuda, 2023.

[15] A. Betts, N. Chong, A. Donaldson, S. Qadeer, and P. Thomson, "GPUVerify: a verifier for GPU kernels," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2012.

[16] G. Li and G. Gopalakrishnan, "Scalable SMT-Based Verification of GPU Kernel Functions," in *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2010.

[17] G. Li, P. Li, G. Sawaya, G. Gopalakrishnan, I. Ghosh, and S. P. Rajan, "GKLEE: Concolic Verification and Test Generation for GPUs," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2012.

[18] A. Amighi, S. Blom, M. Huisman, and M. Zaharieva-Stojanovski, "The VerCors Project: Setting up Basecamp," in *Proceedings of the Sixth Workshop on Programming Languages Meets Program Verification (PLPV)*, 2012.

[19] K. Kojima and A. Igarashi, "A Hoare Logic for GPU Kernels," *ACM Trans. Comput. Logic*, vol. 18, no. 1, 2017.

[20] M. R. Gadelha, F. R. Monteiro, J. Morse, L. C. Cordeiro, B. Fischer, and D. A. Nicole, "ESBMC 5.0: An Industrial-Strength C Model Checker," in *Int. Conf. on Automated Software Engineering (ASE'18)*, 2018.

[21] S. F. Siegel, M. Zheng, Z. Luo, T. K. Zirkel, A. V. Marianiello, J. G. Edenhofner, M. B. Dwyer, and M. S. Rogers, "CIVL: The Concurrency Intermediate Verification Language," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015.

[22] D. Aspetakis. (2023) Ada SPARK GPU Code Examples. https://gitlab.bsc.es/dimitris_aspetakis/ada-spark-gpu.

[23] I. Rodriguez, L. Kosmidis, J. Lachaize, O. Notebaert, and D. Steenari, "GPU4S Bench: Design and Implementation of an Open GPU Benchmarking Suite for Space On-board Processing," Barcelona Supercomputing Center, Tech. Rep. UPC-DAC-RR-CAP-2019-1, 2019.

[24] D. Aspetakis. (2023) GPU4S Bench implementations in Ada SPARK. https://gitlab.bsc.es/dimitris_aspetakis/gpu4s-bench-ada.

[25] D. Steenari, L. Kosmidis, I. Rodriguez-Ferrandez, A. Jover-Alvarez, and K. Förster, "OBPMark (On-Board Processing Benchmarks) – Open Source Computational Performance Benchmarks for Space Applications," in *2nd European Workshop on On-Board Data Processing (OBDP2021)*, 2021.

[26] L. Kosmidis, M. Solé Bonet, I. Rodriguez-Ferrández, J. Wolf, and M. M. Trompouki, "Evaluating the Computational Capabilities of Embedded Multicore and GPU Platforms for On-Board Image Processing," in *European Data Handling and Data Processing Conference for Space (EDHPC)*, 2023.

[27] A. Jover-Alvarez, I. Rodriguez, L. Kosmidis, and D. Steenari, "Space Compression Algorithms Acceleration on Embedded Multi-Core and GPU Platforms," *Ada Lett.*, vol. 42, no. 1, 2022.

[28] D. Steenari et al., "On-Board Processing Benchmarks," 2021, http://obpmark.github.io/.

[29] M. Solé, J. Wolf, I. Rodriguez, A. Jover, M. M. Trompouki, L. Kosmidis, and D. Steenari, "Evaluation of the Multicore Performance Capabilities of the Next Generation Flight Computers," in *Digital Avionics Systems Conference (DASC)*, 2023.